MSc thesis

Computer Science

Making Software Architecture a Continuous Practice

Jacob Lärfors

June 3, 2020

Faculty of Science

University of Helsinki

Supervisor(s)

Prof. Tommi Mikkonen

Examiner(s)

Prof. Antti-Pekka Tuovinen

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Computer Science

Tekijä — Författare — Author

Jacob Lärfors

Työn nimi — Arbetets titel — Title

Making Software Architecture a Continuous Practice

Ohjaajat — Handledare — Supervisors

Prof. Tommi Mikkonen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	June 3, 2020	53 pages

Tiivistelmä — Referat — Abstract

DevOps is an ever growing trend in software development and it conveys a mindset that all things should be *continuous*. Interestingly, one of the common challenges with DevOps adoption is related to software architecture and this is in large due to the fact that architecture is not part of DevOps.

This thesis looks at making software architecture a *continuous* practice and thus bring it into the DevOps space. A prototype solution, Architector, was implemented to solve this and the results indicate that it shows a viable approach to making software architecture a continuous practice. However, further work is necessary to expand the scope of continuous architecture and to fully validate this claim by applying Architector to a real world software development workflow.

ACM Computing Classification System (CCS)
Software and its engineering → Software organization and properties → Software system structures

Avainsanat — Nyckelord — Keywords

DevOps, Software Architecture, Continuous

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software Systems specialisation line

Contents

1 Introduction

DevOps is an ever growing trend in software development and its goal is to minimise the disconnect between development and operations (Smeds et al., 2015). There exists no unified definition of DevOps, which has proven to be one of the reasons teams and organisations struggle with its adoption. Other challenges with adoption have been linked directly to the software architecture, which was a major motivator for this thesis. An interesting observation to consider is that these challenges with adopting DevOps that are related to software architecture only exist because not enough attention is given to architecture (Chen, Babar, and Nuseibeh, 2013).

Agile practitioners have referred to software architecture as Big Up-Front Design (Hasselbring, 2018) DevOps conveys a *continuous* mindset (Fitzgerald and Stol, 2017). Understandably software architecture is not favoured by agile practitioners as traditionally architecture is never changing yet ever present in software architecture, making it a static, non-evolving entity. However, with agile software development software is evolving continuously. Another paradigm which has been observed within the DevOps movement is "everything as code". For example, modern infrastructure provisioning can be automated as it is developed *as code*.

The goal of this thesis is to implement a solution using *architecture as code* that will enable architecture to become a *continuous* practice. A case study will be conducted to apply this solution over an example project.

The method for this thesis is to conduct a study of existing literature to gather the necessary background knowledge of the topic to help drive the solution implementation in the right direction and also to understand the significance of the end result. The background literature is covered in chapter 2.

In chapter 3 some of the available tools and frameworks are studied. The case study is in chapter 4 of which a part was to derive and implement the potential solution which was then applied to an open source project.

In chapter 5 the results of the case study will be presented. Chapter 6 will facilitate the discussion on the results and how effective the proposed solution was in addressing issues discovered by studying background literature. Chapter 7 will conclude the thesis.

2 Background

A study of existing material has been conducted in order to understand the current landscape and relationship between DevOps practices and software architecture. The goal of the research was to understand and provide a clear definition of the terms ”DevOps”, ”Continuous” Practices and ”Software Architecture”. Once the definitions had been established the relationship between DevOps, Continuous Practices and Software Architecture could be analysed, to try and understand how to make software architecture part of the continuous development process, and how this could help alleviate challenges with DevOps adoption

Finally, the research aimed at discovering relationships between DevOps and Software Architecture that had not currently been identified or had not been given significant attention as a way for this thesis to make a contribution.

This chapter is divided up into a several sections to provide a progressive summary of the conducted literature review. Section 2.1 covers existing literature around DevOps in order to provide a definition for this thesis. Section 2.2 studies existing literature around Software Architecture to come up with a sound definition with a bias towards DevOps. Section 2.3 covers existing material on the how architectural decisions can and should be impacted by DevOps practices, with a section on a modern architectural style known as microservices. Section 2.4 looks at the significance and impact software architecture can have on a project, using measures such as Architectural Technical Debt. The final section, 2.5, looks at validating software architectures to try and counter some of the aforementioned Architectural Technical Debt.

2.1 DevOps

The purpose of this section is to introduce and define the concept of DevOps. DevOps has not been given a universally accepted definition, and the lack of a clear definition of DevOps has been observed as one of the challenges with its adoption (Senapathi et al., 2018, Lwakatare et al., 2016). It is therefore important to provide a sound definition for reference in this paper.

Earlier research (Bass et al., 2015) created a definition of DevOps:

> *"DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality"*

Further definitions where made to extend the scope to place more emphasis on the operations side of DevOps (Smeds et al., 2015):

> *"a mindset encouraging cross-functional collaboration between teams - especially development and IT operations - within a software development organization in order to operate resilient systems and accelerate delivery of changes"*

Significant overlap between DevOps and release engineering was perceived, with both having common goals (Smeds et al., 2015). However, inconsistencies of this definition by practitioners led to further research attempting to consolidate a clear understanding of DevOps. The definition provided by (Lwakatare et al., 2016) is based on a scientific definition of DevOps from existing research and further complemented by an exploratory case study. It has been observed that DevOps as a phenomenon has been largely driven and discussed by practitioners, and thus, a lot of information is available on non-scientific forums, typically known as "grey" literature. A Multivocal Literature Review was employed to ensure the full scope of DevOps-related material was observed (Lwakatare et al., 2016). For these reasons, and as no contradiction yet exists for this definition of DevOps, it will be used as the basis for this paper.

The findings from their study (Lwakatare et al., 2016) identified the different dimensions of DevOps as: Collaboration, Automation, Culture, Monitoring and Measuring, which are briefly summarized below.

1. **Collaboration:** Rethinking and reorientation of roles and teams in development and operations activities, so that development teams gain an appreciation and understanding for operations work and vice versa.

2. **Automation:** Infrastructure and deployment process automation using practices like infrastructure-as-code to produce consistent, immutable and scalable environments.

3. **Culture:** Empathy, support and good working environment between development and operations. More frequent and less formal communication between the two groups with shared ownership and responsibility in the case of incidents.
4. **Monitoring:** Instrumenting application and aggregating monitored data into insights.
5. **Measurement:** Useful metrics for both development and operations teams: developers use production feedback and progress is measured in terms of a working system an a production environment.

Given this definition, a large part of DevOps is related to the automated deployment of software that is designed to provide monitoring capabilities of metrics that are useful, and ideally defined by the operations team early on in the process (Lwakatare et al., 2016). Lots of attention has been given to this area of research and less attention has been given to the cultural and collaborative side of DevOps.

Surrounding the DevOps movement are two growing trends. *Continuous* software engineering has been denoted as "Continuous*", the "*" denoting that everything in the software development lifecycle should be continuous and that more continuous practices will arise in the future (Fitzgerald and Stol, 2017). The most well known continuous activities are Continuous Integration, Continuous Delivery and Continuous Deployment. Further continuous activities identified include: Continuous Planning, Continuous Testing, Continuous Compliance, Continuous Security and Continuous Evolution. In this thesis, we are proposing our own continuous activity, *Continuous Architecture*. The second growing trend is *as code*, the most common terms including Infrastructure as Code and Configuration as Code. However, other terms exist like Documentation as Code (for example using Markdown to document your software) and Pipeline as Code (developing your Continuous Integration pipelines using code). In this thesis, we are proposing our own *as code* term, *Architecture as Code*. It is noteworthy that not much literature exists around the *as code* paradigm yet it is a very common industry term, and similarly to *Continuous Everything*, there is a growing trend of *Everything as Code*.

2.2 Software Architecture

This thesis is focusing on the relevance of DevOps within the scope of software architecture and this section introduces a basic definition of architecture based on existing literature.

IEEE Standard 1471 documents a consensus on good architectural description practices (Maier et al., 2001). IEEE 1471 defines architecture as:

> *The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.*

One point that this definition highlights is the difference between an architectural description and an architecture. In short, architectural descriptions can be considered a collection of artifacts or documents that describe a system's architecture, where as an architecture is more of a concept or attributes of a system.

A software architecture discussion group at SEI in 1994, prior to the IEEE 1471 standard, derived a general definition (Garlan and Perry, 1995):

> *The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.*

As far as general purpose definitions go, this definition is pretty useful in explaining what software architecture refers to, even a few decades later. However, there is further clarification needed to help explain the various views that one might have regarding software architecture. For instance, when discussing software architecture people may refer to (Garlan and Perry, 1995):

- the high-level architectural layout of the system, describing each component and its purpose/function
- architectural styles that may have been followed to influence the high-level architectural structure
- specific coding and design guidelines used to accomplish consistency across the developed system
- the general study of software architecture from an academic standpoint

Their research (Garlan and Perry, 1995) went on to identify two trends emerging in the software architecture field. The first was the recognition of designers facing similar challenges and successes and thus deriving and contributing towards a repertoire of methods,

models, patterns and idioms for describing, communicating and structuring complex software systems.

The second trend was the recognition of domain specific requirements, and the development and reuse of frameworks for product families. Typical examples include; standardized communication protocols, high level user interface libraries and fourth-generation programming languages built on abstractions of reuse.

Interestingly, we can observe that these architectural trends still hold true and especially within the field of DevOps. For example, more recent work about software architecture for enabling Continuous Delivery (Chen, 2015) outlines a number of Architecturally Significant Requirements (ASRs), derived from challenges, contributing towards a repertoire of DevOps adoption techniques and thus confirming the first trend. The second trend can be confirmed by emerging architectural styles, such as microservices, which have been influenced by Domain Driven Designs (Shadija et al., 2017) and have been observed as a key architectural style towards enabling DevOps (Chen, 2018).

2.3 Architecting for DevOps

Given the earlier defined understanding of DevOps, several papers make explicit claims about the impact software architecture has in DevOps adoption (e.g. Chen, 2015, Chen, 2018, Bellomo et al., 2014, Shahin et al., 2016). It has been recognized that insufficient attention is given to software architecture in the topic of Continuous Delivery (Chen, 2018), a practice essential in DevOps adoption. Architecturally Significant Requirements were introduced as a way to bring architectural design decisions into the requirements phase (Chen, Babar, and Nuseibeh, 2013), making architecture a topic for requirements such as deployability, modifiability and testability, which are all heavily influenced by architecture. Microservices have become an increasingly popular architectural style that has shown to help with DevOps adoption because it remediates a number of the pains encountered in larger monolithic software systems. One study (Shahin et al., 2016) concluded that the architectural challenge that received most attention when adopting Continuous Deployment, was the existence of highly coupled monolithic systems, especially when multiple teams are all dependent on the same codebase and making unrelated changes at the same time. One developers change could affect the deployability of the entire codebase, making it difficult for others to test their new features. This kind of scenario is typical in layered architectures and is therefore not a concern when developing microservices. This isola-

tion of development is nothing new with microservices, just that microservices makes this separation explicit whereas layered system designs have a tendency to erode and leave a highly-coupled monolithic system that is difficult to test, modify and maintain.

There is significant research in the field of DevOps that references the microservices architectural style (e.g. Chen, 2018, Shadija et al., 2017, Balalaie et al., 2016). In order to make use of the literature around DevOps and microservices it seemed necessary to explore the existing definitions of microservices. Similarly to the definitions of DevOps and Software Architecture, no globally accepted definition of microservices exists today. One recent study (Shadija et al., 2017) aimed to provide a clear understanding of microservices. Their study compared microservice-style architectures with traditional Service Oriented Architectures (SOA) such as a layered or multi-tier architecture. The main difference is that with layered or multi-tier architectures a single feature typically touches the whole system as that feature likely needs functionality in the Presentation, Business, Data, and any other relevant layer. In comparison, microservices are single, independently deployable units of code modelled around business capabilities. Thus, in a microservices-style architecture a feature might be only coupled by the UI and a single microservice.

2.4 Significance of Continuous Software Architecture

The purpose of this section was to research the impact, experience and significance of software architecture in real projects to quantify the potential value of making software architecture a *continuous* practice. A term known as *Technical Debt* is a metaphor used to describe the deficiencies in quality of software systems. Research (Nord et al., 2012, Martini and Bosch, 2015) has make direct ties between Technical Debt and the architecture of a system, referring to it as *Architectural Technical Debt*, describing technical debt as a continuous compromise between short-term and long-term goals, which is particularly interesting in the area of architecture with its strong ties to software maintenance. The topic of technical debt is especially relevant in agile development where decisions are made continuously and frequently, given that the core principle of technical debt is the accumulation of interest on any debt incurred. Increase the scope from enterprise agile development to the DevOps space where quality attributes like deployability, testability and monitorability become core quality attributes, the technical debt of architecture becomes increasingly important.

Knowing that technical debt is a continuous trade-off that is made with every decision and

that software architecture is a main contributor to technical debt, the lack of architecture-related tasks in agile practices (such as scrum) is clearly identified in one analysis (Nord et al., 2012). Their work went on to conclude that the scope of agile practices, which typically focuses on short-term value of features, must be widened before one can consider the consequences on the long-term success of a software system. It is therefore imperative to provide visibility into the existing architecture (and its current technical debt) in order to make informed decisions about short-term versus long-term trade-offs. In essence, making software architecture part of the *continuous* software development workflow through continuous monitoring of the software architecture.

More recent work went on to categorise Architectural Technical Debt and provide an indication of the impact or cost associated with the debt (Martini and Bosch, 2015). Their taxonomy included the following five Architectural Technical Debt items:

1. **Dependency violations and unawareness** are related to the presence of undesired architectural dependencies that violate the design principles of the software architecture. This type of architectural technical debt was mentioned in every case investigated (Martini and Bosch, 2015).

2. **Non-uniformity of patterns and policies** includes general design patterns that should be made consistent throughout the project, such as handling communication, or coding guidelines.

3. **Code duplication (i.e. non-reuse)** relates to the presence of duplicate (or similar) code in different parts of a system. This leads to maintenance overhead and the opportunity for bugs to get introduced.

4. **Temporal properties of inter-dependent resources** are *hidden* dependencies that are represented in the temporal space, such as communication through some bus, where two parts of code (e.g. client and server) are dependent on each other.

5. **Unidentified non-functional requirements** refers to non-functional requirements that were not included in the architectural design phase and can be thus, incredibly difficult to recover from as significant changes might need to be made.

What is interesting from their case study (Martini and Bosch, 2015) is the estimated impact these types of technical debt have on a system. To help identify architectural technical debt items that incur significant interest through a series of knock-on effects,

they introduced the notion of *vicious circles*. The highlighted vicious circles in their research include:

1. **Contagious Architectural Technical Debt** describes the kind of technical debt which has a sequence of knock-on effects that causes interest to accumulate in the effected parts. Hence the technical debt cannot be measured only by the initial source of the infection (or technical debt).

2. **Hidden Architectural Technical Debt, Incomplete Refactoring and Time Pressure** are three items that have been combined under the idea that hidden technical debt cannot be calculated and therefore leads to incomplete refactoring efforts to pay of incurred debt, which is further strained by time pressures to deliver new software releases.

3. **Lack of uniformity and familiarity** leads to greater accumulation of technical debt. This is regarded as a vicious circle if this particular combination is met, and would therefore lead to the continuous growth of technical debt

The vicious circles explained above are essentially a result caused by the accumulation of architectural technical debt items, based on the aforementioned taxonomy.

Other research has studied how software architecture fits in with agile software development. Some research claims that there is significant friction between the agile software development community and software architecture, with the former referring to architecture work as Big Up-Front Design (BUFD) (Hasselbring, 2018). Their paper goes on to the future of software architecture where the goal should be to include continuous architecture validation into the agile software development process, which forces continuous architectural decisions to be made and keeps architecture as a focus point of the development process. A major concern for agile development, as identified earlier, is the lack of architectural focus. This idea of continuous validation would be a significant contribution, given that even the best software architecture is worthless if the code does not conform to the architecture (Clements and Shaw, 2009). Then there are topics of Architectural Technical Debt studied earlier and the understanding that is needed for developers to work with software systems. This means it should be a two-way relationship: if the software conforms to the intended architecture design it will increase the overall quality of the software, and if the architecture design matches the software it will help knowledge sharing and developing the software further. Other studies have been made discussing the

significance of keeping the documentation and knowledge about the software up to date to avoid architectural erosion (Goltz et al., 2015).

With agile practitioners referring to software architecture work as Big Up-Front Design (Hasselbring, 2018), one interesting case study focused on whether a satisfactory software architecture would emerge as part of continuous refactoring during an agile software project (Chen and Babar, 2014) without the need for Big Up-Front designs and continuous architecture validation. They conducted a survey with 102 practitioners and their results were 20 contextual factors that affect whether a satisfactory architecture emerges. Many of these factors would greatly benefit by improving communication and architectural understanding and awareness. Continuous Integration was stated as an influencing factor, with governance (or validation) of the architecture proving useful. Common challenges like culture, team composition, geographical location, management and test automation were all factors that came out of the study. Their results indicate that there are clear benefits to integrating software architecture into the continuous software development process, for example through Continuous Integration and architectural validation.

Thus, many studies have highlighted the benefits to bringing software architecture into the continuous software development workflow (through practices such as Continuous Integration), yet there is a considerable lack of research on how this can really be achieved. This would not just mean validating architectural compliance but creating a process in such a way that developers are considerate and aware of architecture and are involved in making architectural decisions ensuring that the architectural designs provide value in describing and communicating the system, as well as minimising any possible architectural erosion (deviation from the intended design).

The categories of architectural technical debt explained in this section, together with the identification of viscous circles, provides a great foundation for targeting architectural analysis against. To properly understand which parts of architectural technical debt can be minimized a study of existing architectural models is required. If models are used to explain how the software should be built, and which dependencies are desired and undesired, then bring validation of those designs into an agile development process will be possible.

2.5 Modelling Software Architecture

Modelling software architecture has been noted by many authors problematic (Kruchten, 1995) because typically one diagram cannot represent or describe the architecture of a system. Sometimes too much detail is added to one diagram or it is not clear what boxes and arrows mean: are the boxes components and the arrows data, control or environmental dependencies? As such, a *4+1 View Model* was developed to remedy the problem, which describes 5 different concurrent views of an architectural description (Kruchten, 1995):

1. The **Logical View** is primarily concerned with the functionality that the system provides to the end-users. Typically Unified Modelling Language (UML) diagrams are used to represent the logical view.

2. The **Process View** describes the system's dynamic aspects, such as concurrency and synchronisation, including how system processes communicate.

3. The **Physical View** illustrates the software and how this maps onto the hardware, also considered the deployment view.

4. The **Development View** describes the software's layout and organisation, usually modelled with UML using diagrams such as component and package diagrams.

5. The **Scenario View** is the "+1" in the model, and is concerned with use cases and scenarios using architectural descriptions from the other views.

There are several technologies available for modelling software architecture, and these technologies also provide several different models for creating different architectural viewpoints. Among them, the Unified Modelling Language (UML) is considered as the most used language by practitioners for modelling software architectures (Ozkaya and Erata, 2020). Thus, to help limit the scope of the work this thesis will consider only UML as the standard for modelling software architecture viewpoints because of its popularity and range of different models.

UML offers two categories of diagrams, under which every type of UML diagram belongs to; structural (or static) and behavioural (or dynamic) diagrams. Structural diagrams are concerned with the system's (static) structure, of which the following are examples: composite structure diagram, deployment diagram, class diagram and component diagram. Behavioural diagrams are concerned with the system's changing (dynamic) behaviours,

and examples are: use case diagram, state machine diagram, and various interaction diagrams that describe communication and timing of a system.

A recent study (Ozkaya and Erata, 2020) conducted a survey with 109 practitioners about UML diagrams, based on category, and which are the most commonly used diagrams. They categorised their UML diagrams into the following, based on the model's viewpoint: Functional, Information, Concurrency, Development, Deployment and Operational. They further divided these viewpoints into model types, for which specific UML diagrams could be used for the design. Their survey asked participants to list the UML diagrams they use for each of these model types.

For the scope of this case study the focus is on the *Development View* (as defined in the 4+1 Model) using the relevant UML diagram types for which inspiration can be taken from the case study by Ozkaya and Erata, 2020 to create an architectural description. The following UML diagrams were used by participants in the survey in this case study: Profile Diagram, Package Diagram, Object Diagram, Deployment Diagram, Composite Structure Diagram, Component Diagram and Class Diagram. The most popular diagrams for modelling the software module structure were Package Diagram, Component Diagram and Deployment Diagram. The most popular diagrams for modelling the source-code structure were Package Diagram, Class Diagram and Component Diagram.

During the case study it will be necessary to create a few different views of the software architecture using these structural UML diagrams and find one that enables an architect to adequately describe the architecture for a development team to implement the software, and a piece of software to read the design, derive rules and check whether the software conforms to the intended design. Hence, when the tooling landscape is considered it is crucial that the available tools can support these different types of UML models.

2.6 Validating Software Architecture

Validating software architecture can either refer to validating the software architectural design (e.g. Selonen and Xu, 2003), or validating that the software conforms to the intended architecture (e.g. Cheon et al., 2009, Jiang et al., 2007, Aldrich, 2008, Rudzki et al., 2008, Kiviluoma et al., 2006). For the purpose of this thesis we are studying the latter. Reasons for validating that software conforms to its intended design include; ensuring that essential quality attributes are achieved (Aldrich, 2008), that Applicable Programming Interfaces (APIs) available to developers are used correctly (Jiang et al., 2007), ensuring

the software system does not suffer from design drift or architectural corrosion (Cheon et al., 2009, Rudzki et al., 2008) which generally makes communication of the software architecture more difficult across teams or multi-site development locations.

Several approaches to this have been suggested, mostly using static analysis techniques and some using dynamic analysis techniques. Of the static types, one very promising approach (Rudzki et al., 2008) wanted to tackle the problem of multi-site development and automate the process of architectural enforcement to inform developers of design drift. Their approach involved creating an XML file to describe the intended architecture that could be understood by their validation tool and validate the software architecture. One interesting thought that this paper provokes is how the XML file is maintained and ensuring that design drift is not introduced between the actual software design and the XML file used for automatic validation. Furthermore, the paper allowed for wild-cards to enable system evolution. However, this could lead to errors if the wild-cards are used too permissively and thus lead to False-Negatives (the model incorrectly assesses the architecture).

Another promising approach (Aldrich, 2008) made three technical contributions to the field:

1. A precise definition of *communication integrity*.
2. Reasoning about communication through shared data, yielding a static approach to enforcing architecture of a run-time (dynamic) component and connector view of architecture for object-oriented programs.
3. Demonstration of architectural constraints that directly contribute to quality attributes such as performance and extensibility

.

The work describes *communication integrity* as critical to yielding benefits built into the architecture (such as performance and extensibility). By their definition, two components communicate whenever:

1. **Direct call:** Component instance `A` or an object in one of its ownership domains invokes a method directly on component instance `B`, or
2. **Connection call:** Component instance `A` invokes a method of component instance `B` through a connection, or

3. **Shared data:** An object in architectural domain `A` *accesses* (invokes a method or reads or writes a field of) a non-component object `B` which is in a different architectural domain

.

This provides a clear definition of the types of checking that are required for a tool to enforce an architectural design of a traditional system (e.g. where the dependencies are explicit in the software).

Both of the above approaches have demonstrated the possibility to statically enforce design intent on a software system. However, they have required some intermediate step going from a software architecture design to enforcement. The first approach (Rudzki et al., 2008) used an intermediary XML file, and the second approach (Aldrich, 2008) required the development of the architectural design and compilation using the ArchJava toolkit in order to enforce it.

Of the dynamic or run-time architecture enforcement approaches, most approaches monitored the execution of a system to report inconsistencies with the described architecture. One approach (Kiviluoma et al., 2006) used *profiles* described by UML models to generate Java AspectJ code that is used to monitor the execution of the program code. This approach could be mostly automated as the UML models are part of the architectural design, which is used to automatically generate the AspectJ code. Using this approach, quality attributes (such as those needed for Continuous Delivery and DevOps-style practices) that are expressed in the architecture, which if modelled using UML, could be achieved automatically through run-time monitoring. Similar more recent work (Cheon et al., 2009) extended these capabilities by using OCL to declaratively describe constraints over the UML model, meaning that architectural ”rules” are described in OCL.

One paper suggested using run-time monitoring to create documentation to aid developers in correctly using APIs (Jiang et al., 2007). This approach did not check for inconsistencies of a program at run-time but helped aid developers so that APIs would not be misused. The output was sequence diagrams to help developers understand the flow of a program and how it interacted internally.

3 Tooling

The goal of this chapter is to research the available tools for modelling software architecture *as code* and analysing software architecture by parsing source code. This chapter is split into two sections. Section 3.1 looks at tools for modelling architecture diagrams, and section 3.2 looks at tools for analysing the architecture of source code.

3.1 Architecture Modelling Tools

One of the core concepts of the proposed Continuous Architecture method is to achieve modelling *as code* so that the designs can be stored in a source code repository (such as Git) together with the source code, and be parsed or interpreted easily to derive design rules, and hence only textual modelling tools were considered. Additionally, as this process should be able to integrate into a pipeline and the model to serve as documentation, any tool that is only available online without an easy API to generate and embed images was disregarded. In the following section the tools that currently support the above requirements have been listed. In order to find a list of textual modelling tools, a series of Google searches were performed.

PlantUML (*PlantUML* n.d.) is one of the most well-known textual modelling tools with an eco-system of integrations and plugins. It is written in Java and can be executed from the command line (using a distributed JAR). There is also an online playground provided for quick prototyping. PlantUML supports many different UML diagrams, including Component and Class diagrams, making it a candidate for this case study.

One really great advantage to PlantUML is its ecosystem, for example integrating with Visual Studio Code (an extremely popular text editor) as well as Git server providers such as GitHub and GitLab allowing you to embed PlantUML into Markdown.

As a very brief example, the following code renders the a sequence diagram as shown in Figure 3.1.

```
@startuml
Alice -> Bob: Authentication Request
Bob --> Alice: Authentication Response
```

```
Alice -> Bob: Another authentication Request
Alice <-- Bob: Another authentication Response
@enduml
```

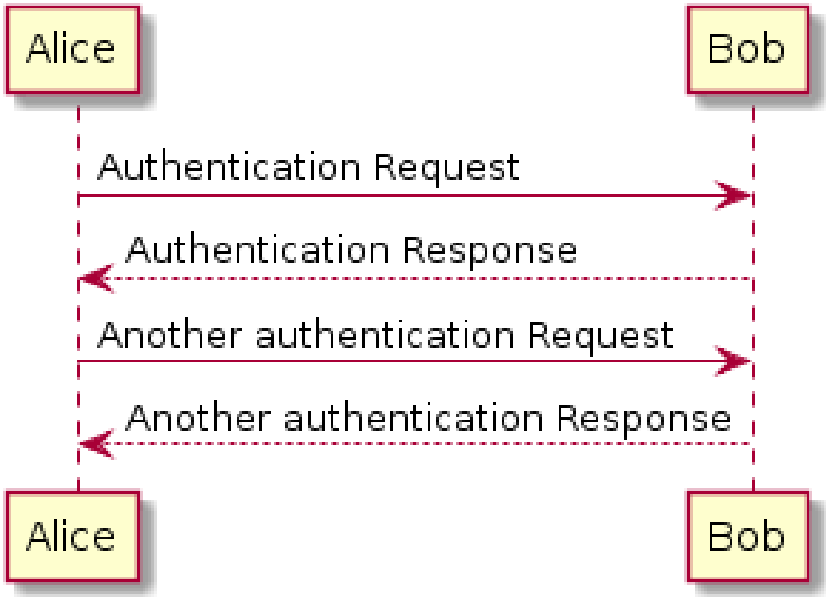

Figure 3.1: PlantUML Example Diagram.

Nomnoml (*Nomnoml* n.d.) is a more recent textual UML modelling tool written in Javascript using Nodejs. Its industry usage and ecosystem does not seem as well developed as that of PlantUML, nor does it specifically list the types of UML diagrams that it supports, but it does have a comprehensive syntax allowing one to create a plethora of different UML diagrams.

Nomnoml provides a command line interface meaning that it can be integrated into an automated process and does not require any service to be running separately.

An example taken from the Nomnoml website can be seen in Figure 3.2, which is produced by the following code snippet.

```
[Pirate|eyeCount: Int|raid();pillage()|
  [beard]--[parrot]
  [beard]-:>[foul mouth]
]
```

```
[<abstract>Marauder]<:--[Pirate]
[Pirate]- 0..7[mischief]
[jollyness]->[Pirate]
[jollyness]->[rum]
[jollyness]->[singing]
[Pirate]-> *[rum|tastiness: Int|swig()]
[Pirate]->[singing]
[singing]<->[rum]
```

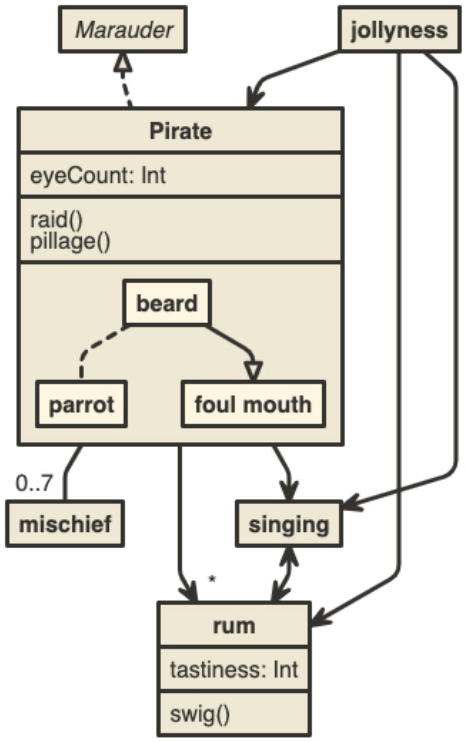

Figure 3.2: Nomnoml Example Diagram.

There seem to be several other textual UML modelling tools, and most were not considered because of dependencies on legacy looking interfaces (meaning restrictions on automation), lack of UML diagram support, commercial licenses and general lack of activity surrounding the project. Here is a list of textual modelling tools that looked promising but were disregarded:

1. **yUML** was not considered because it appears to be commercial and only available

as on online service, unless you are an enterprise customer. Further, it seemed to support a limited set of UML diagrams

2. **TextUML** provides a very vast range of different UML diagrams, but requires an online editor or a special IDE to be used to create the models. Therefore it was not considered

3. **ZenUML** was not considered because it only seems to support sequence diagrams

Given the two candidates, more in-depth research was needed with practical application between PlantUML and Nomnoml. Given the Google Trend result in Figure 3.3 comparing PlantUML and Nomnoml, it is quite clear which tool is more popular.

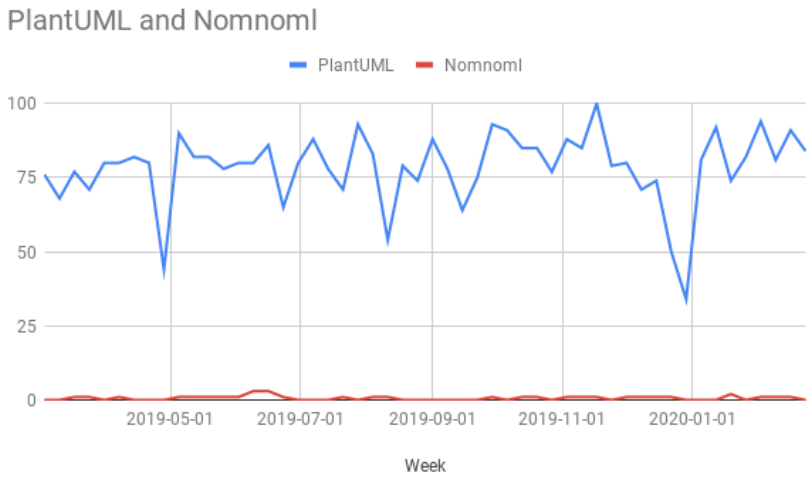

Figure 3.3: Google Trends: PlantUML vs Nomnolm, past 12 months, Worldwide

Based on the requirements for a modelling tool and the wide industry use of PlantUML, it was decided that PlantUML would be used as the modelling tool.

3.2 Architecture Analysis Tools

In order to understand the architecture of source code a architectural analysis tool is needed. For the scope of this thesis it was decided to only look at static architectural analysis tools, and not consider those architectural analysis tools that use dynamic analysis (such as monitoring an application to deduce the architecture).

For the scope of this case study and thesis it was decided to focus on only explicit code dependencies, such as function calls, and ignore any dependencies in the temporal space,

such as communication via APIs or some serial bus. Furthermore, it was decided to limit the example language to C (or C++) to make the selection of an example project and parser more straight forward.

When looking for a parser for C or C++ code that can retrieve dependencies in the codebase there are a number of static analysis tools and frameworks available, of which three were considered for our purposes.

1. **Lattix** (*Lattix Website* n.d.) is a commercial tool that provides a Dependency Structure Matrix for source code and enables the setting of rules for which dependency violations can be exported

2. **Understand by SciTools** (*Understand by SciTools* n.d.) is a commercial tool that provides insight and understanding of a project code base, with the ability to extract dependency information via diagrams or an API, yet no direct functionality for rules

3. **Clang** (*Clang Tooling Documentation* n.d.) an open source compiler that provides front-end libraries to inspect the generated Abstract Syntax Tree which contains information about elements and their dependencies

Based on previous experience and the availability of Clang (that is, no license is needed) it was decided that Clang, specifically the LibClang library, would be used as the code parsing tool.

4 Case Study

The goal of this case study is to implement a prototype solution that enables *Continuous Architecture* using *Architecture as Code* to model the software architecture. This chapter includes the case study that was conducted on the open source project IsoAgLib. The chapter is divided into seven sections. Section 4.1 describes the motivation of the case study. Section 4.2 outlines the approach to architectural modelling *as code*. Section 4.3 outlines the approach to parsing source code and understanding the architecture. Section 4.4 describes how architectural violations were calculated. Section 4.5 covers the example project IsoAgLib, why it was chosen and how it was used in the case study. Section 4.6 discusses the general implementation of the prototype solution, called Architector. Section 4.7 documents how to use the command-line interface of Architector.

4.1 Motivation

Continuous Architecture will help bring software architecture into a continuous software development process using continuous DevOps-style practices such as Continuous Integration. The core idea behind this solution is to use *architecture as code* enabling the model of the architecture to be used to design, describe and also enforce the software architecture. As the architecture model is described using code which resides together with the project codebase (for example, in a markdown file) it follows the same development workflow that the source code follows (it is reviewed by peers regularly), and is used in Continuous Integration to check that the actual source code conforms to the designed architecture.

In order to conduct this case study a prototype tool needed to be implemented which was named Architector for reference throughout this thesis and was written in Python. Architector should scan a project code base and understand the architecture (dependencies of the codebase) and compare this to an architecture model, written as code, describing the design of the software. Figure 4.1 shows an overview of Architector's process. The Architectural Compliance process is the main function that Architector performs and outputs a report of all the architectural violations, which are instances of the source code not conforming to the architecture described in the architecture model.

There are three basic components to this:

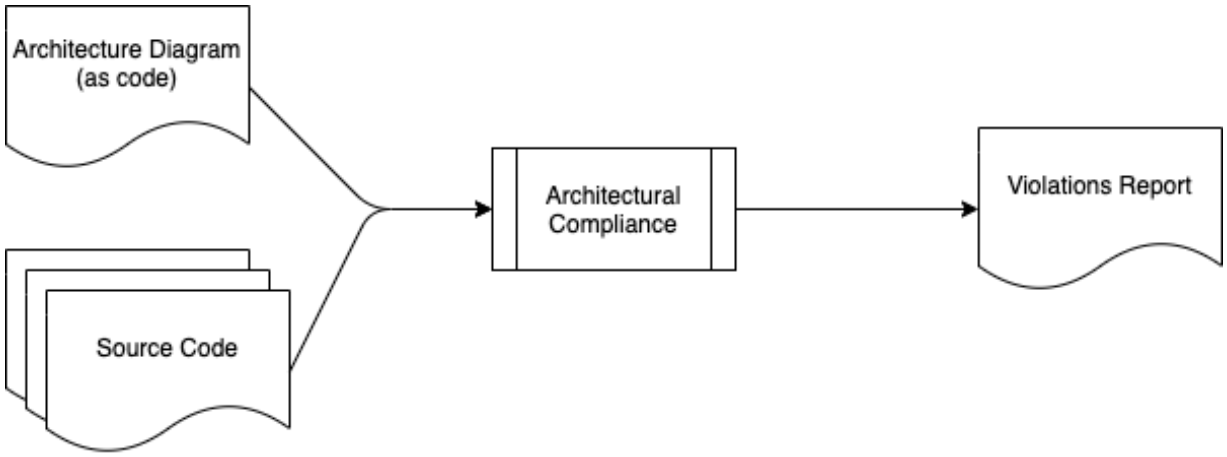

Figure 4.1: Architector Process Overview

1. Parsing an architecture diagram and understanding dependencies.
2. Parsing source code and understanding source code dependencies.
3. An engine that compares the desired architectural dependencies with the actual source code dependencies.

To narrow the scope of this case study it was decided to make a clear and narrow definition of software architecture dependencies and thereby what an architectural violation is. For the purpose of this case study an **Architectural Dependency** is considered a **Direct Call**, as defined in (Aldrich, 2008) based on the **Development View** in the 4+1 Model (Kruchten, 1995). Hence, an architectural dependency is when component instance A or an object of its ownership domain invokes a method directly on component instance B. To clarify, this means transient calls, or the use of shared or global data and communication via APIs (such as in microservices) will not be considered in this case study. This means only explicit direct code dependencies are considered.

For the definition of an **Architectural Violation** the definition from (Martini and Bosch, 2015) is used which defines **Dependency Violations** as the presence undesired architectural dependencies. Thus, an instance of an undesired architectural dependency is considered an architectural violation. An architectural dependency is considered undesirable if the architectural model does not describe the dependency.

4.2 Architectural Modelling as Code

Of the available UML diagrams in PlantUML the Component Diagram was considered the best fit for this use case as it can be closely compared to the folder structure and design of source code. Notably this would not be the only diagram that would be used in a project but is one that most closely aligns with the scope of this case study.

4.2.1 PlantUML: Component Diagram

The main items available in PlantUML's Component Diagrams are described below with some potential use cases for the Architector implementation.

1. **Components** which can correspond to a source code file or set of files (such as a source and header file in C/C++)
2. **Interfaces** which could be used as a subset of components, specifically to denote interfaces between components
3. **Packages** which are used to group components and can be used to denote the directory structure for an application

Based on the available items it was decided to simplify the diagrams and use only Components and Packages for this case study.

4.2.2 PlantUML: Parsing Model Code

In order to parse and understand dependencies within a PlantUML model the underlying language was studied. PlantUML does not use a formal grammar for parsing its language but uses raw regular expressions. Hence, an engine to parse the PlantUML language needed to be created and for this it was decided to write a grammar for a subset of the PlantUML language.

Most of the PlantUML examples that can be found online use a syntax similar to the example in Listing 4.1 for writing models

Listing 4.1: Calculator PlantUML Code

```
@startuml
```

```
[My First Component] as comp1
[Another component] as comp2
@enduml
```

To make writing the grammar easier and make developing PlantUML diagrams stricter it was decided to write PlantUML models using the example syntax in Listing 4.2.

Listing 4.2: Calculator PlantUML Code

```
@startuml
component "My First Component" as comp1
component "Another Component" as comp2
@enduml
```

Based on research of different ways to write grammars the Extended Backus-Naur Form (EBNF) was chosen to develop a grammar to parse PlantUML diagrams using this restricted subset of the PlantUML language. To ensure the language subset of PlantUML was sufficient to describe a software architecture a simple PlantUML test was developed, based on a command line calculator application.

The PlantUML code in Listing 4.3 was able to generate the architecture model in 4.2

Listing 4.3: Calculator PlantUML Code

```
@startuml calc-diagram

package "app" as p_app {
  component "app" as c_app
  component "calc" as c_calc
}

package "lib" as p_lib {
  component "math" as c_math
  component "print" as c_print
}
c_app -> c_calc
c_app --> c_print

c_calc --> c_math
```

```
@enduml
```

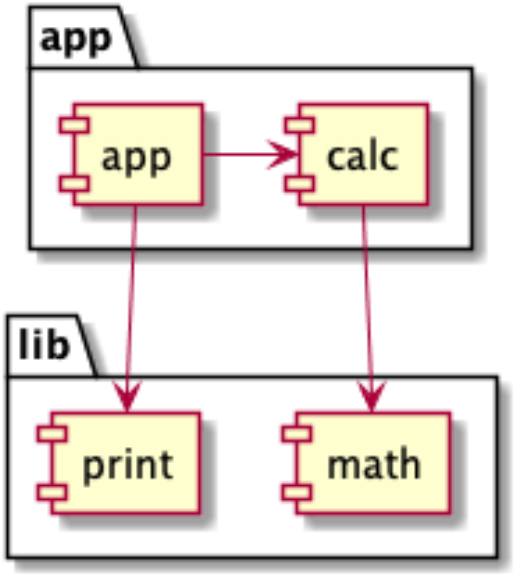

Figure 4.2: Calculator PlantUML Component Diagram

The component diagram for the example calculator application, Figure 4.2, describes four basic components and their relationships.

1. **app** uses calc and print directly, but should not depend on math directly
2. **print** is used by app and nothing else
3. **calc** is used by app and depends on math
4. **math** is used by calc and nothing else

If a developer of the calculator application were to introduce an unwanted dependency, such as the calc module directly invoking the print module, or app calling math directly, Architector should be able to report violations of any such instances based on the PlantUML diagram in Listing 4.3

Using the EBNF notation the grammar in Listing 4.4 was developed to parse the subset of PlantUML code described in Listing 4.2.

Listing 4.4: EBNF Gramma for PlantUML Subset

```
start: "@startuml" title? (entity | relationship)* "@enduml"

title : VARIABLE

entity : package
       | component

package : "package" name "as" variable body? stereotype?
component : "component" name "as" variable body? stereotype?

body : "{" entity* "}"

stereotype : "<<" VARIABLE ("/" VARIABLE)* ">>"

relationship : VARIABLE (DEP_USES | DEP_USED_BY) VARIABLE

name : ESCAPED_STRING

variable : VARIABLE

DEP_USES : /-{1,2}>/
DEP_USED_BY : /<-{1,2}/

VARIABLE : /[A-Za-z0-9-_]+/

COMMENT: "'" /[^\n]/*
%ignore COMMENT

%import common.ESCAPED_STRING
%import common.WS

%ignore WS
```

4.3 Parsing Source Code Dependencies

Clang provides a number of interfaces to inspect source code (*Clang Tooling Documentation* n.d.), of which LibClang is the most suitable for this case study because it provides the most stable and high-level interface with Python bindings. In order to inspect source code using LibTooling a Translation Unit needs to be created which requires the arguments to compile each source file for a project. An easy approach to this was to use a Compilation Database JSON file, see example in Listing 4.5, which contains the following three elements for each source file:

1. **Command:** the command line invocation to compile the source file.
2. **Directory:** the working directory from which the compilation command is run.
3. **File:** the source file that is being compiled.

Listing 4.5: Example Compilation Database JSON File

```
[
   {
       "directory": "/home/user/llvm/build",
       "command": "clang++ -c -o file.o file.cc",
       "file": "file.cc"
   }
]
```

Compilation Databases have become somewhat standard within the C and C++ space and there are several ways to create them. The most straight forward is if the CMake build tool is being used then a `compile_commands.json` file containing a list of compile commands for each source file in the project can be easily generated. LibTooling has a utility class called CompilationDatabase that can be used to interact with a compile commands JSON file.

The LibTooling interface provides cursors to traverse the Abstract Syntax Tree (AST) of a Translation Unit. A translation unit is generated from an entry in the compilation database for a given source file. LibTooling provides a lot of detailed semantic information about elements in the AST of which a few we are interested in. One very convenient method is `clang_getCursorReferenced()` which returns a cursor to a referenced entity,

or `NULL` if there is no reference. Using this interface method it is straight forward to retrieve the explicit dependencies in a translation unit.

For the implementation of Architector using libclang, a *source code dependency* is a dependency from any *source* entity within the code base on any *target* entity within the code base, unless:

1. The *source* entity exists in side a system header (e.g. `#include <system.h>`).
2. The *source* entity exists in the same file as the *target* entity (because of the file index these dependencies do not need to be considered).
3. The *source* entity is a preprocessing directive (such as a macro), except for includes. Inclusion directives are considered by Architector.

This is made simple as LibClang has an enum called `CXCursorKind`. Every cursor (consider this a source code entity) in LibClang has a cursor kind, and this makes it trivial to filter or include cursors. The enum range `40 <= cursor_kind <= 50` was used by Architector to include all cursors of type *reference* in a non-expression context (e.g. TypeRef, NamespaceRef, MemberRef, LabelRef, VariableRef). The enum range `101 <= cursor_kind <= 105` includes all expressions that *reference* other entities.

4.4 Calculating Architectural Violations in Source Code

Understanding dependencies in PlantUML models and in the source code enables a comparison against the two to identify violations of the architecture in the source code. An Architectural Violation is defined as a dependency in the source code that is not described as a relationship in the PlantUML Component Diagram. Thus, every source code dependency must be *allowed* by the PlantUML component diagram. This makes discovering architectural violations quite straightforward: every source code dependency is a violation unless the PlantUML component diagram has a relationship between the source and destination nodes.

Every dependency has a source node and a destination node. In the context of an AST, a source node might be a function call and the destination node would be the declaration of the called function in a header file. See the short snippets of code in Listings 4.6 and 4.7, and based on the component diagram in Figure 4.2.

Listing 4.6: Source File: app/app.h

```
#include "lib/print.h"
```

Listing 4.7: Source File: app/calc.c

```
... // other things here
#include "app.h"
...
```

Based on this example, Architector should be able to observe that that `app.h` includes `print.h` from the `lib` module, and by `calc.c` including `app.h` it is inherently dependent on `print.h` which violates the intended architecture as described in Figure 4.2.

For the purpose of this case study dependency violations will be calculated on a file level. That means the smallest entity to be described in a PlantUML component diagram should be a file, and the comparison of dependencies between a component diagram and source code should be performed on a file level. Therefore, to make the calculation of architectural violations a `File Index` was created which is an index for every file and directory in the source code. This means every source file will have a unique index and can be used to reference a dependency as a 2-dimensional index `(x, y)` where `x` is the source dependency file index, and `y` is the destination dependency file index. See Listing 4.8 for an example file index for the simple calculator example given the component diagram in Figure 4.2.

Listing 4.8: Source File: app/calc.c

```
0: app/app.c
1: app/app.h
2: app/calc.c
3: app/calc.h
4: lib/math.c
5: lib/math.h
6: lib/print.c
7: lib/print.h
```

Using the example file index from Listing 4.8 and the component diagram in Figure 4.2 it is quite straight forward to derive the allowed architectural dependencies using 2-dimensional indices. Considering only at the relationship between the components `app` and `calc`, where `app --> calc`, this can be defined using the following indices `(0, 2), (0, 3), (1, 2),`

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 1 & 0 & 4 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad Z = AB = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 4.3: Calculating Architectural Violations using File Index Matrices

`(1, 3)`. This set of indices means both `app.c` and `app.h`, can use both `calc.c` and `calc.h`.

Using indices makes it very easy to create matrices, and matrices are easy to compare in order to find violations. The approach taken by Architector is there to create a matrix based on the PlantUML component diagram and a matrix based on the source code dependencies, using a common file index to compare the two. The matrix corresponding to the component diagram is initialised with ones, and for each pair of indices the value is set to zero. The matrix corresponding to the source code takes the opposite approach and is initialised with zeroes, and for each pair of indices the matrix index value is incremented by 1. When the matrices are multiplied all non-zero values in the output matrix are violations with their number of instances. Consider the file index in Listing 4.8 using again the subset of components `app` and `calc`, where `app --> calc`. Let us denote the matrix corresponding to the component diagram as A, the matrix corresponding to the source code B, and the output matrix is Z. See the equation in Figure 4.3 showing the output matrix Z with a non-zero index of `(2, 1)` which based the the file index means `calc.c` depends on `app.h`. There are some implicit dependencies in matrix A such that source files and header files can be dependent on each other without having to be explicitly denoted in a component diagram.

4.5 Example Project

To conduct the case study a suitable example project is needed. In order to analyse the source code dependencies access to the source code is needed and therefore choosing an open source project made sense. Based on earlier decisions to use Clang as the front-end source code parser the example project needed to be written in C or C++. Furthermore, a project from which a suitable architecture could be obtained and developed in PlantUML would make the analysis possible.

Several open source projects were considered based on searches on GitHub. Industry known projects such as Git, Vim and cURL were considered but their directory structure did not describe any form of architecture, and thus deriving any kind of architecture model from the project was not straightforward, if at all, possible.

IsoAgLib (*IsoAgLib* n.d.) is an open source library for handling communications within embedded software according to ISO 11783. Conveniently, the project provides a system architecture overview as seen in Figure 4.4 which is one of the main reasons why it was chosen for this case study.

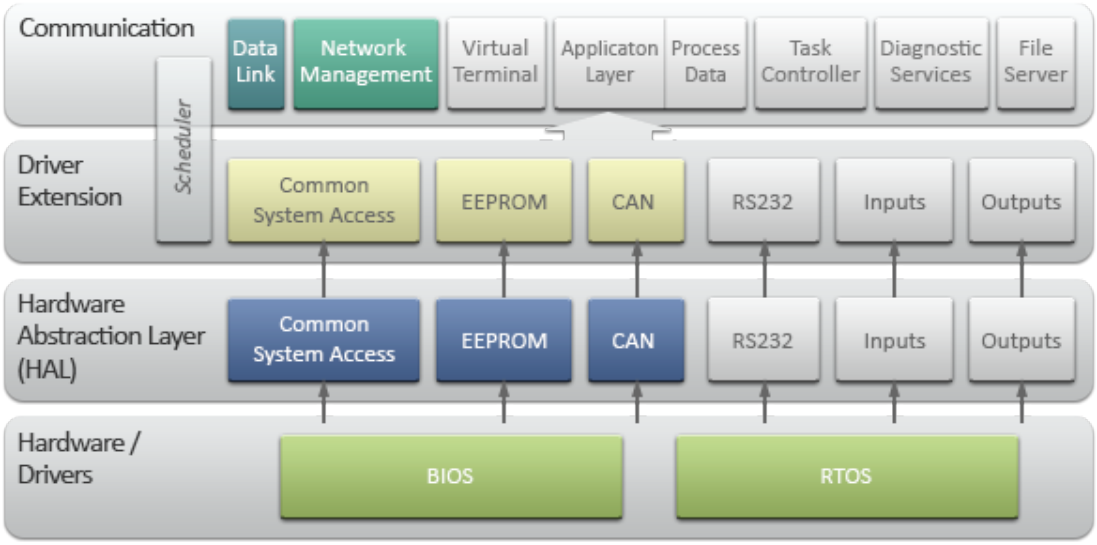

Figure 4.4: IsoAgLib System Architecture

Based on this high-level system architecture and cross-referencing to the source code, a PlantUML model was developed using Listing 4.9 which produces the corresponding diagram as seen in Figure 4.4.

Listing 4.9: IsoAgLib PlantUML Architecture

```
@startuml IsoAgLib

package "library" as library {
  package "xgpl_src" as xgpl_src {
    package "IsoAgLib" as isoaglib {
      package "comm" as comm {
        component "Part3_DataLink" as comm_data_link
        component "Part3_ProprietaryMessages" as comm_prop_msg
```

```
        component "Part5_NetworkManagement" as comm_net_mgmt
        component "Part6_VirtualTerminal_Client" as comm_vtc
        component "Part7_ApplicationLayer" as comm_app
        component "Part10_TaskController_Client" as comm_task_ctl
        component "Part12_DiagnosticsServices" as comm_diag_svc
        component "Part13_FileServer_Client" as comm_file_srv
      }
      package "scheduler" as scheduler {

      }
      package "driver" as driver_ext {
        component "can" as driver_ext_can
        component "system" as driver_ext_system
      }
      package "hal" as hal
      package "util" as util
    }
    package "supplementary_driver" as supp_driver {
      component "driver" as supp_driver_driver
      component "hal" as supp_driver_hal
    }
  }
}

comm <-> scheduler
comm --> driver_ext

scheduler <--> driver_ext

driver_ext --> hal

hal --> util

@enduml
```

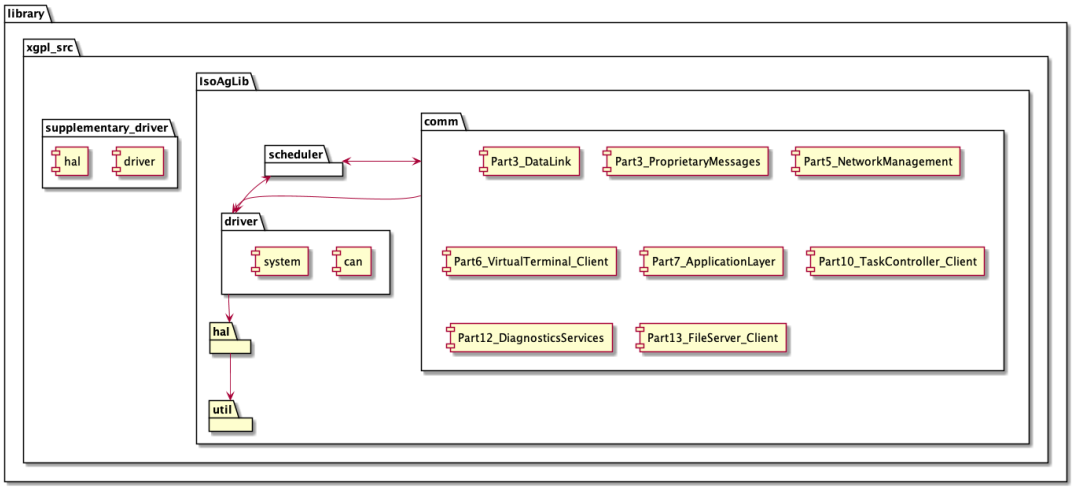

Figure 4.5: IsoAgLib PlantUML Architecture

Comparing the diagrams in Figures 4.4 and 4.5 one can see that a number of assumptions had to be made. For example, the IsoAgLib System Architecture in Figure 4.4 uses data dependencies to illustrate the direction of dependencies (the arrows point upwards) instead of control dependencies. The assumption was made that the architecture would follow that of a typical Layered Architecture where upper dependencies can use the lower dependencies. It is also very noticeable that there is not a clear one-to-one mapping of components in the system architecture diagram (Figure 4.4) to the components in the directory structure and PlantUML diagram (Figure 4.5) and this highlights the disconnect between the intended software architecture and the actual software architecture. Studying a little further the differences, a number of the components within the Communication layer are present in the directory structure however their names contain additional data providing a part number. Additionally, in the system architecture (Figure 4.4) the "Application Layer" and "Process Data" components within the Communication layer are grouped together, yet only the "Application Layer" exists as a directory called "Part7_ApplicationLayer".

Interestingly all the studied example projects have either a clear lack of architectural intent in their folder structure, or if there is an architectural diagram or description, the difference between intended architecture and structure of the codebase is significant. This makes creating diagrams using PlantUML that have to map to a file index (that is, the directory and file structure) challenging.

4.6 Implementation of Architector

Given an example project, IsoAgLib, a parser able to identify source code dependencies, Clang, and an architectural model *as code* using PlantUML, these had to be tied together. For prototyping purposes Python was chosen as the programming language using Pipenv (*Pipenv* n.d.) to manage the environment and dependencies. Some of the notable dependencies are:

1. **fire** is used to create a minimal command line interface,
2. **lark-parser** is used to parse the EBNF grammar for parsing PlantUML model diagrams,
3. **libclang** is used to parse the C++ source code,
4. **numpy** is used for the matrix calculations,
5. **joblib** is used to parallelise the parsing of source code for efficiency.

In the beginning of this chapter Figure 4.1 described on a high level the process of Architector. Figure 4.6 shows a more detailed process diagram of Architector. Below is a description of the different components in this more detailed diagram.

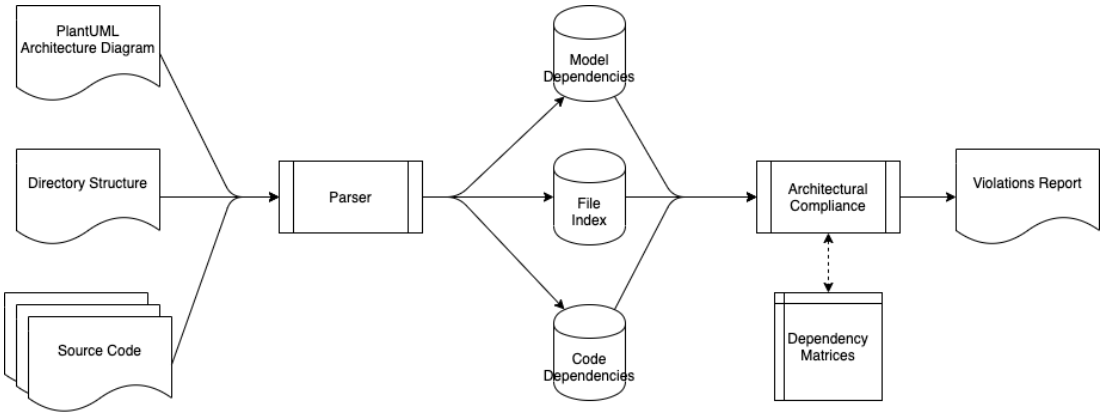

Figure 4.6: Architector Process in Detail

Inputs to the process include:

1. A PlantUML diagram (as code) describing the intended architecture of the software

2. The directory structure from which to form the file index from. This could simply be a base directory from which to walk and process each file and directory

3. The source code together with a specification of how and what to parse. In this case study this would include the Compile Commands JSON file which Clang requires to parse the code. IsoAgLib uses CMake for its build process which means extracting a compile commands JSON file was straight forward by providing `-DCMAKE_EXPORT_COMPILE_COMMANDS=1` as an extra argument to the `cmake` command

Parser takes the inputs and produces three different output data types. For prototyping purposes a Python dictionary was used to store the data. The three output data types are:

1. **Model Dependencies** are the dependencies that are described in the *architecture as code* diagram using PlantUML. The dependency source and destination entities reference the file index.

2. **File Index** stores the list of files and directories collected from a given base directory and provides the index for the model dependencies and the source code dependencies

3. **Code Dependencies** contains all of the dependencies obtained from parsing the source code. Similarly to the Model Dependencies, the source and destination entities for each dependency references the file index so that the source code and model dependencies can be compared in a dependency matrix

The **Architectural Compliance** process takes the File Index, Model Dependencies and Code Dependencies and creates the dependency matrices in-memory (earlier described in Figure 4.3) in order to calculate the architectural violations and outputs a report. For simplicity, the report file is in a Comma-Separated Values (CSV) format with the columns `source,destination` where the source and destination specify the source file which is retrieved from the file index. It is worth noting that the specific source code atoms are stored in the Code Dependencies data store and could be retrieved so that the report could contain also the line number and source range of the atom (that is, the specific location in the source file of the source and destination dependencies).

Listing 4.10 shows an example snippet from the violations report. In this example you can see that the source dependency is coming from supplementary_driver component and is referencing the communication component, which is supposed to be uppermost layer in the architectural model.

Listing 4.10: Example Violations Report for IsoAgLib

```
/library/xgpl_src/supplementary_driver/driver/eeprom/impl/eepromio_c.h,/
    ↪ library/xgpl_src/IsoAgLib/comm/Part5_NetworkManagement/impl/
    ↪ isoname_c.h
```

4.7 Using the Architector Command Line Interface

Architector was implemented as a Command Line Interface (CLI) tool so that it can be easily integrated with in Continuous Integration system as part of an automated development set-up. The Python library `fire` makes this incredibly straight forward by parsing a Python class and making the methods exposed as command line arguments.

Listing 4.11 shows the `main.py` entry point to the program, and Listing 4.12 shows a stripped down version of the ArchAnalyzer class with the method analyse which is used to invoke the analyser.

Listing 4.11: Architector Source File: main.py

```
from fire import Fire

from lib.arch import ArchAnalyzer

if __name__ == '__main__':
        Fire(ArchAnalyzer)
```

Listing 4.12: Architector Source File: arch.py

```
class ArchAnalyzer:
        def analyze(self, puml_file: str, compdb: str, base_dir: str = "."
            ↪ ):
                # 1. Calculate the file index using base_dir
                # 2. Parse the PlantUML diagram specified with puml_file
                # 3. Parse the source code specified by the compilation
                    ↪ database compdb
                # 4. Calculate and output the architectural dependencies
```

Architector can then be invoked from the command line as shown in Listing 4.13

Listing 4.13: Example Command Line Usage of Architector

```
# first install the dependencies specified in the Pipfile
$ pipenv install

# invoke architector's analyze command
pipenv run python main.py analyze \
       --base_dir /path/to/isoaglib \
       # the path to the PlantUML file which is usually given
       # the suffix .puml
       --puml_file /path/to/isoaglib/isoaglib.puml \
       # typically the build directory where cmake is run from
       # contains the compile_commands.json file
       --compdb /path/to/isoaglib/build
```

By invoking Architector in this way, the example output shown in Listing 4.14 is shown on the command line.

Listing 4.14: Architector Example Output

```
$ pipenv run python main.py analyze --base_dir ../isoaglib/ --puml_file
   ↪ ../isoaglib/isoaglib.puml --compdb ../isoaglib
Loading .env environment variables...
2020-05-23 13:15:01,950 - lib.fs - INFO - Scanning file system: Starting
   ↪ ...
2020-05-23 13:15:02,009 - lib.fs - INFO - Scanning file system: Done!
2020-05-23 13:15:02,009 - lib.puml - INFO - Processing PlantUML File:
   ↪ Starting...
2020-05-23 13:15:02,009 - lib.puml - INFO - Parsing PlantUML File:
   ↪ Starting...
2020-05-23 13:15:02,053 - lib.puml - INFO - Parsing PlantUML File: Done!
2020-05-23 13:15:02,056 - lib.puml - INFO - Processing PlantUML File: Done
   ↪ !
2020-05-23 13:15:02,056 - lib.code - INFO - Parsing C/C++ Project:
   ↪ Starting...
2020-05-23 13:15:02,056 - lib.code - DEBUG - Creating compilation database
   ↪  from directory ../isoaglib using 8 cores
Parsing source file [2/128]: /Users/jacoblarfors/work/uni/thesis/isoaglib/
```

```
   ↪ library/xgpl_src/supplementary_driver/driver/datastreams/
   ↪ filestreamoutput_c.cpp
Parsing source file [1/128]: /Users/jacoblarfors/work/uni/thesis/isoaglib/
   ↪ library/xgpl_src/supplementary_driver/driver/datastreams/
   ↪ filestreaminput_c.cpp
...
...
Parsing source file [127/128]: /Users/jacoblarfors/work/uni/thesis/
   ↪ isoaglib/library/xgpl_src/supplementary_driver/hal/pc/rs232/
   ↪ target_extension_rs232_simulating.cpp
Parsing source file [128/128]: /Users/jacoblarfors/work/uni/thesis/
   ↪ isoaglib/library/xgpl_src/supplementary_driver/hal/pc/eeprom/
   ↪ target_extension_eeprom_simulating.cpp
2020-05-23 13:16:16,701 - lib.code - INFO - Parsing C/C++ Project: Done
TOTAL VIOLATIONS = 1628
```

5 Results

The goal of the case study was to develop a prototype solution, which was named Architector, that would enable software architecture to become a continuous process. This chapter covers the results from the case study, and is divided into two sections. The section 5.1 describes the capabilities of Architector, and the section 5.2 covers how Architector was applied to IsoAgLib and the results that were produced. Overall the implementation of Architector was successful as it delivered the high-level capabilities detailed in the following sections.

5.1 Architector Capabilities

This section discusses Architector's capabilities with two high-level themes: architecture *as code* and architectural violations.

5.1.1 Architecture as Code

One goal of Architector was to enable a single source of truth for the software architecture design that would provide a reference for both the intended and actual software architecture. A result of Architector is that the intended software architecture (that is, the architectural design) is the same as the actual software architecture of the source code by reporting architectural violations. Architectural violations are instances were the actual software architecture does not conform to the intended software architecture.

The intended software architecture was developed using PlantUML which provides a language for developing UML diagrams. For the case study, component diagrams were chosen as the type of UML diagram to provide the software architecture design. PlantUML Component Diagrams allow software engineers to describe the different source code entities (for example source files, components, modules, packages) and how the entities interact with each other, which is their relationship and dependencies on one another.

5.1.2 Architectural Violations

Given an input PlantUML architecture model (as code) and the corresponding source code, Architector is able to compare the intended architectural dependencies with the actual source code dependencies. Architector is able to deduce architectural rules from the PlantUML model that specify the dependencies that are allowed or intended. Architector is able to parse C or C++ source to obtain the actual dependencies in the source code. Finally, Architector is able to compare the intended and actual dependencies to produce a list of architectural violations.

The comparison of architectural dependencies is performed using a file index which means the lowest level of architectural violations are on a source file level. This means that Architector does not support architectural rules on source code entities within a source file (e.g. on a class level), but this was considered as a reasonable limitation for the scope of this case study.

5.2 Applying Architector on IsoAgLib

The example project chosen for the case study was IsoAgLib. The goal was to apply Architector to this project and study how feasible the application of Architector was on a real project.

5.2.1 IsoAgLib: Architecture as Code

The implementation of IsoAgLib's architecture as code diagram is shown in Figure 4.5 which is generated from the PlantUML Component Diagram code in Listing 4.9. The model was based on the IsoAgLib system architecture in Figure 4.4 and cross referencing the source code structure. One immediate challenge was that the source code structure did not reflect the system architecture and some assumptions were needed. A further challenge came with *implicit* dependencies, such as a module depending on itself. By default if component A is dependent on itself, and this is not described in the intended architecture, it is treated as an architectural violation. For the scope of this project these implicit dependencies were added to the PlantUML Component Diagram which made the generated UML diagrams more explicit but also more complicated to comprehend. Another type of implicit dependencies is concerning layers in a layered architecture. Consider

the system architecture for IsoAgLib, an assumption was made that the Communication layer can directly depend on any lower level layers (this is due to the significant number of dependencies on lower level layers). This includes Communication being able to depend on the Hardware / Drivers layer (the lowest level layer in the IsoAgLib system architecture diagram in Figure 4.4).

Figure 5.1 shows the component diagram that was generated using Listing 5.1 which contains all of the implicit dependencies that were assumed in the IsoAgLib source code. This includes the Communication layer being able to talk with every other component. It is worth noting that dependencies for the sub-components within the Communication layer were not specified as they were inherited by the Communication layer dependencies. With a deeper understanding of a project it would be prudent to try and specify more thoroughly the architectural intent rather than specifying dependencies on the highest level layers. For the purpose of this case study and not being overly familiar with the IsoAgLib codebase it made sense to base the dependencies on the layers in the system architecture diagram (Figure 4.4 on page 30).

Listing 5.1: IsoAgLib PlantUML Architecture with Implicit Dependencies

```
@startuml IsoAgLib

package "library" as library {
  package "xgpl_src" as xgpl_src {
    package "IsoAgLib" as isoaglib {
      package "comm" as comm {
        component "Part3_DataLink" as comm_data_link
        component "Part3_ProprietaryMessages" as comm_prop_msg
        component "Part5_NetworkManagement" as comm_net_mgmt
        component "Part6_VirtualTerminal_Client" as comm_vtc
        component "Part7_ApplicationLayer" as comm_app
        component "Part10_TaskController_Client" as comm_task_ctl
        component "Part12_DiagnosticsServices" as comm_diag_svc
        component "Part13_FileServer_Client" as comm_file_srv
      }
      package "scheduler" as scheduler {

      }
```

```
      package "driver" as driver_ext {
        component "can" as driver_ext_can
        component "system" as driver_ext_system
      }
      package "hal" as hal
      package "util" as util
    }
    package "supplementary_driver" as supp_driver {
      component "driver" as supp_driver_driver
      component "hal" as supp_driver_hal
    }
  }
}

comm -> comm
comm -> scheduler
comm --> driver_ext
comm --> hal
comm --> util
comm --> supp_driver

scheduler -> scheduler
scheduler -> comm
scheduler --> driver_ext
scheduler --> hal
scheduler --> util

driver_ext --> driver_ext
driver_ext --> scheduler
driver_ext --> hal
driver_ext --> util

hal --> hal
hal --> util
```

```
util --> util

supp_driver --> supp_driver
supp_driver --> util

@enduml
```

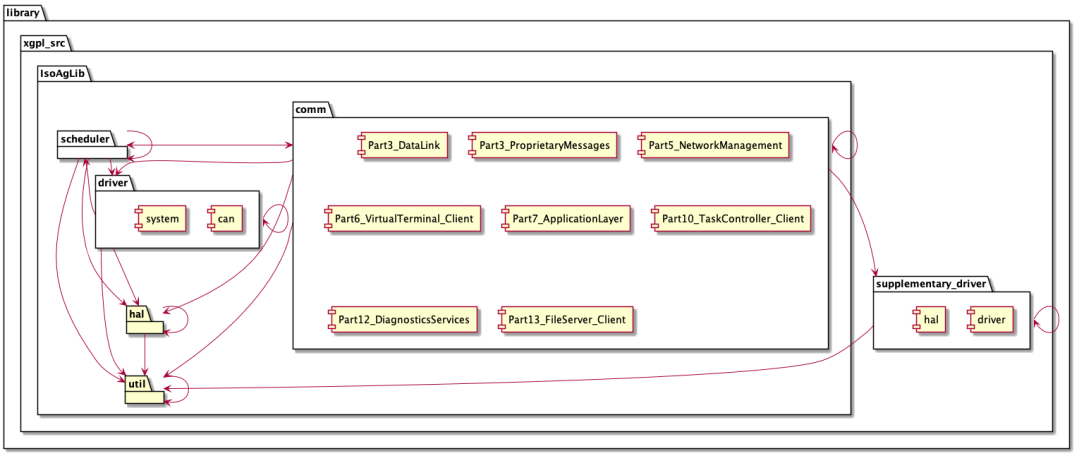

Figure 5.1: IsoAgLib PlantUML Architecture

5.2.2 IsoAgLib: Architectural Violations

Using the PlantUML component diagram in Listing 5.1 and running it over release version 2.9.2 of IsoAgLib, the total number of reported violations was 1628. Figure 5.2 shows the number of architectural violations by component.

Table 5.1 shows the breakdown of the architectural violations by source and destination components. The largest number of architectural violations between components is from `driver` to `comm`, with a significant number of violations from `hal` to `driver` and `util` to `supplementary_driver`.

What is interesting from these results is that all of the dependencies from `driver` on `comm` are from a single header file which includes different parts of `comm`. This likely indicates that either the intended system architecture was not considered when implementing `driver`, or the system architecture does not adequately describe the intended architecture. Either way, this is a significant form of architectural technical debt. As `comm` is the highest level layer in IsoAgLib it has dependencies on most of the rest of the library. Referring

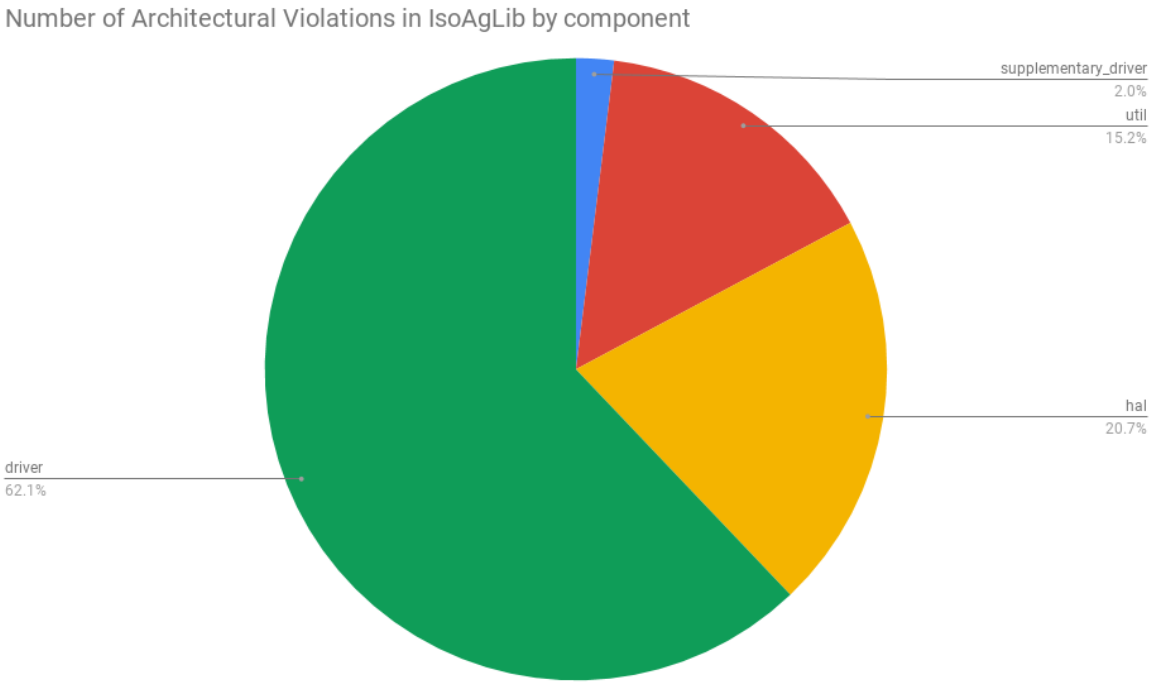

Figure 5.2: IsoAgLib Architectural Violations by Component

to the definition of vicious circles in (Martini and Bosch, 2015), this is a typical form of Contagious Architectural Technical Debt.

Source	Destination	# violations
driver	comm	1011
hal	driver	298
util	supplementary_driver	248
hal	comm	36
supplementary_driver	hal	22
supplementary_driver	comm	10
hal	scheduler	3

Table 5.1: IsoAgLib Architectural Violations by Component

6 Discussion

The goal of this thesis was to conduct a case study of a prototype tool for making software architecture a *continuous* practice. This section is broken down into two subsections: firstly an analysis of the results relating back to the background study that was conducted, and secondly threats to validity.

6.1 Analysis

The scope of this case study was very limited focusing only on a subset of software architecture, considering only direct calls within the software architecture description *development view* (as per the 4+1 Model). Whilst the results are still relevant, this leaves a considerable gap for other parts of software architecture to be made continuous, such as analysing data dependencies, considering deployment architectures (such as dependencies on file system), compliance to coding guidelines and patterns, and any other relevant properties relating to software architecture. Nonetheless, to help evaluate the success of Architector towards *Continuous Architecture*, given the scope of this thesis, it is worthwhile to consider briefly a typical software development workflow.

Consider a typical development workflow (such as trunk based development) where developers create branches from a trunk branch in a version control tool such as Git, and then create Pull Requests (or Merge Requests) when these changes are ready to be merged back to the trunk. In this scenario the pull request facilitates a quality gate that includes a continuous integration pipeline running followed by a peer code review. In order for the code to be merged both the continuous integration pipeline and the code review need to be approved. Part of the continuous integration pipeline would include invoking the Architector command line to produce a list of architectural violations. If violations are reported, the developers would be aware that the architecture has changed and therefore needs to be reviewed. There are two possible scenarios:

1. The developer has violated the architectural design. The fix is to change the code and comply with the software architecture

2. The software has evolved beyond the last revision of the architecture, and the soft-

ware architecture design needs to be updated to reflect new development efforts (such as a new feature or a new module being developed)

In the former case, the developers can modify the source code. In the latter case, it is the software architecture model, as code, that would need to be modified and these changes would become part of the code review, which in turn need to be approved by peers and possibly a technical lead. As the software architecture is modelled *as code* it will follow the same workflow and evolve together with the source code and be continuously validated against the source code and continuously reviewed.

Some interesting further observations to make would be non-technical factors. The definition of DevOps comprises a number of different dimensions. Architector would fall deeply into the Automation dimension by providing another level of automation in Continuous Integration pipeline and providing feedback on the software architecture. However, as software architecture becomes a continuous practice, the affect on the DevOps dimensions of Collaboration and Culture would be very interesting to explore. To consider that software architecture has traditionally been a static process performed by a technical lead or software architect, the review and update of an architectural design will need to be considered as an on going activity. A positive outcome of this would be that architecture becomes a very prevalent topic which helps overcome to observed lack of focus that it gets in software development (Chen, 2018). It might also help solve the question of continuous architectural refactoring in an agile project, and whether this yields a satisfactory architecture (Chen and Babar, 2014). Instead of continuously refactoring the software architecture, the focus would be towards continuously reviewing and designing the architecture because the compliance to that design is built into the continuous development process. The effects of this could be significant as it would force the design to be part of the development process. However, one could argue that it may stagnate development if the design tasks become to large. For example, software architecture has been referred to as Big Up-Front Design (Hasselbring, 2018) and one would need to make sure that continuous architecture does not become Continuous Big Up-Front Design.

Another interesting topic to study is about the adoption of Architector. If it is brought in at the beginning of a project the adoption might be quite smooth because there is no Architectural Technical Debt. However, if Architector is adopted in the middle of a project there is likely already some Architectural Technical Debt that would need to be considered. The two clear choices would be whether to design the ideal software architecture with PlantUML and accept that many architectural violations exist, or design an architecture

that reflects the current software and make an active attempt to refactor the architecture in parts, therefore limiting the amount of architectural violations.

Having analysed the viability of Architector as an approach to making architecture a continuous practice it is worth analysing the significance of the results with regards to DevOps adoption. One study from the background chapter (Shahin et al., 2016) found that the biggest challenge in adoption of Continuous Deployment (a fundamental practice in DevOps) are highly-coupled monolithic systems. Highly coupled systems are created when insufficient attention is paid to the architecture and how different parts of a system become inter-dependent on each other. If architecture becomes part of the development and on going design process of a system, it is likely to raise concerns about how coupled a system is. Engineers would not design a system to be highly coupled, and thus if the design is continuously checked against the actual system source code, it would highlight the architectural validations and provide a means to avoid a system from becoming highly coupled. Furthermore, if architectural design becomes a common discussion and topic, perhaps it would enable development teams to avoid building large monolithic applications by better structuring their components and modules in a more logical way.

Other studies (Nord et al., 2012, Martini and Bosch, 2015) highlighted the connection between technical debt and the software architecture of a system, describing technical debt as a continuous compromise between short-term and long-term goals. The bias that agile software development brings towards short-term goals could be balanced when considering the software architecture and ensuring long-term quality attributes are achieved, including quality attributes that better enable DevOps and its inherent practices (like Test Automation, Continuous Deployment).

Considering some non-technical factors, two studies (Clements and Shaw, 2009, Goltz et al., 2015) have mentioned the value of a software architecture design when it comes to communication and collaboration. The design has been described as worthless if it does not reflect the real software, and how documentation of the architecture helps avoid architectural erosion. Architector would solve this challenge by ensuring that the software does conform to the design, and forces the existence of the design to evolve together with the software. An interesting thought in the realm of DevOps is to consider how useful an completely trustworthy architectural design document would be for an operations team to maintain and monitor an application that has been developed by a development team. One would assume that such a document would greatly benefit the operations team in understanding how things fit together and provide a high-level overview of the software

composition.

From this analysis there are two very relevant ideas for future work:

1. Expanding the scope of Architector to support more than explicit direct calls in the software architecture to include more dynamic properties, such as data dependencies and microservice dependencies

2. Conduct a further case study adopting Architector in a real software development project to analyse the adoption and what the change in workflow is and how that impacts the project from a collaborative and cultural perspective, not just a technical one

6.2 Threats to Validity

There are two topics covered as potential threats to the validity of this research:

1. Are the results from Architector correct and relevant?

2. Does Architector provide a viable approach for architecture to become a continuous practice?

Of the 1628 violations produced by Architector against the IsoAgLib example project, only 10 of them were checked manually, and the rest were verified based on their component dependencies. Of the 10 manually inspected violations they were all correct, based on the PlantUML component diagram. It would be necessary to study the source code of IsoAgLib to verify more of the architectural violations and ensure these are correctly reported and that there are no False Positives. If one assumes the results are of Architector are correct, then the relevance of issues is entirely dependent on the design of the component diagram UML model that is created.

If the results from Architector are correct and relevant, does it provide a viable approach to making software architecture continuous? Given the example workflow in the analysis subsection of the discussion, it would be fair to say that Architector has presented a viable approach for making software architecture part of the continuous development process by enabling the architecture to evolve together with the source code and remaining a prevalent topic. However, to fully validate this claim, especially considering non-technical factors such as collaboration and culture, it would be necessary to conduct a further case study,

bringing Architector into a real software development workflow and observing its adoption and potential issues that could arise.

7 Conclusion

The goal of this thesis was to implement a prototype solution that could enable software architecture to become a continuous practice. In the background chapter, relevant literature was studied around DevOps and continuous practices, and in the tooling chapter some research of existing tools and frameworks was performed to have the relevant information to conduct the case study. In the case study Architector was used to produce architectural violations against a PlantUML component diagram of the example project IsoAgLib, of which the outcome was reported in the results chapter. In the discussion chapter an analysis of the results was discussed, and an example workflow was considered that would enable Architector to be integrated into a software development workflow. A subsection of the discussion was to delve into potential threats to the validity of this research.

In conclusion, for the scope of this research Architector has proven to be a viable approach to automatically validating software architecture against an architectural design, and forcing the existing of such an architectural design to evolve together with the source code. This removes the possibility of any disconnect between the design of the software architecture and the actual software architecture. When architectural violations are reported it was discussed that two options are available: either fix the source code to conform with the software architecture or update the software architecture design. This enables software architecture to become a topic of continuous discussion, and makes it part of the software development process. Making it part of the software development process means collaboration of the architecture is needed and automation is used to enforce the architecture. By the definition of DevOps and continuous practices, this indicates that Architector would help make architecture a continuous practice. However further research is necessary to understand what the extent of this means, and how significant the impact would be to the team adopting Architector.

Bibliography

Aldrich, J. (2008). "Using types to enforce architectural structure". In: *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pp. 211–220.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2016). "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". In: *IEEE Software* 33.3, pp. 42–52. ISSN: 0740-7459. DOI: 10.1109/MS.2016.64.

Bass, L., Weber, I., and Zhu, L. (2015). *DevOps: A software architect's perspective*. Addison-Wesley Professional.

Bellomo, S., Ernst, N., Nord, R., and Kazman, R. (2014). "Toward design decisions to enable deployability: Empirical study of three projects reaching for the continuous delivery holy grail". In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 702–707.

Chen, L. (2015). "Towards architecting for continuous delivery". In: *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, pp. 131–134.

– (2018). "Microservices: architecting for continuous delivery and DevOps". In: *2018 IEEE International Conference on Software Architecture (ICSA)*, pp. 39–397.

Chen, L. and Babar, M. A. (2014). "Towards an evidence-based understanding of emergence of architecture through continuous refactoring in agile software development". In: *2014 IEEE/IFIP Conference on Software Architecture*, pp. 195–204.

Chen, L., Babar, M. A., and Nuseibeh, B. (2013). "Characterizing architecturally significant requirements". In: *IEEE software* 30.2, pp. 38–45.

Cheon, Y., Avila, C., Roach, S., and Munoz, C. (2009). "Checking design constraints at run-time using OCL and AspectJ". In: *International Journal of Software Engineering* 3.1, pp. 5–28.

Clang Tooling Documentation (n.d.). URL: https://clang.llvm.org/docs/Tooling.html.

Clements, P. and Shaw, M. (2009). ""The Golden Age of Software Architecture" Revisited". In: *IEEE software* 26.4, pp. 70–72.

Fitzgerald, B. and Stol, K.-J. (2017). "Continuous software engineering: A roadmap and agenda". In: *Journal of Systems and Software* 123, pp. 176–189.

Garlan, D. and Perry, D. E. (1995). "Introduction to the special issue on software architecture". In: *IEEE Trans. Software Eng.* 21.4, pp. 269–274.

Goltz, U., Reussner, R. H., Goedicke, M., Hasselbring, W., Märtin, L., and Vogel-Heuser, B. (2015). "Design for future: managed software evolution". In: *Computer Science-Research and Development* 30.3-4, pp. 321–331.

Hasselbring, W. (2018). "Software architecture: Past, present, future". In: *The Essence of Software Engineering*. Springer, Cham, pp. 169–184.

IsoAgLib (n.d.). URL: https://isoaglib.com/en/home.

Jiang, J., Koskinen, J., Ruokonen, A., and Systa, T. (2007). "Constructing usage scenarios for API redocumentation". In: *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pp. 259–264.

Kiviluoma, K., Koskinen, J., and Mikkonen, T. (2006). "Run-time monitoring of architecturally significant behaviors using behavioral profiles and aspects". In: *Proceedings of the 2006 international symposium on Software testing and analysis*, pp. 181–190.

Kruchten, P. B. (1995). "The 4+ 1 view model of architecture". In: *IEEE software* 12.6, pp. 42–50.

Lattix Website (n.d.). URL: https://www.lattix.com/.

Lwakatare, L. E., Kuvaja, P., and Oivo, M. (2016). *An Exploratory Study of DevOps: Extending the Dimensions of DevOps with Practices*.

Maier, M. W., Emery, D., and Hilliard, R. (2001). "Software architecture: Introducing IEEE standard 1471". In: *Computer* 34.4, pp. 107–109.

Martini, A. and Bosch, J. (2015). "The danger of architectural technical debt: Contagious debt and vicious circles". In: *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pp. 1–10.

Nomnoml (n.d.). URL: http://www.nomnoml.com/.

Nord, R. L., Ozkaya, I., Kruchten, P., and Gonzalez-Rojas, M. (2012). "In search of a metric for managing architectural technical debt". In: *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pp. 91–100.

Ozkaya, M. and Erata, F. (2020). "A Survey on the Practical Use of UML for Different Software Architecture Viewpoints". In: *Information and Software Technology*, p. 106275.

Pipenv (n.d.). URL: https://pipenv.pypa.io/en/latest/.

PlantUML (n.d.). URL: https://plantuml.com/.

Rudzki, J., Hammouda, I., and Mikkonen, T. (2008). "Ensuring architecture conventions in multi-site development". In: *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pp. 339–346.

Selonen, P. and Xu, J. (2003). "Validating UML models against architectural profiles". In: *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 58–67.

Senapathi, M., Buchan, J., and Osman, H. (2018). "DevOps Capabilities, Practices, and Challenges: Insights from a Case Study". In: *Proceedings of the 22Nd International Conference on Evaluation and Assessment in Software Engineering 2018*. EASE'18. New York, NY, USA: ACM, pp. 57–67. ISBN: 978-1-4503-6403-4. DOI: 10.1145/3210459.3210465. URL: http://doi.acm.org/10.1145/3210459.3210465.

Shadija, D., Rezai, M., and Hill, R. (2017). "Towards an understanding of microservices". In: *2017 23rd International Conference on Automation and Computing (ICAC)*, pp. 1–6.

Shahin, M., Babar, M. A., and Zhu, L. (2016). "The intersection of continuous deployment and architecting process: practitioners' perspectives". In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 1–10.

Smeds, J., Nybom, K., and Porres, I. (2015). "DevOps: a definition and perceived adoption impediments". In: *International Conference on Agile Software Development*, pp. 166–177.

Understand by SciTools (n.d.). URL: https://scitools.com/.